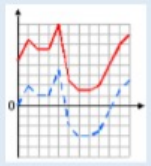
SKLEARN TRANFORMERS – PART 1

- FOR SCALING & CENTERING DATA FEATURES -

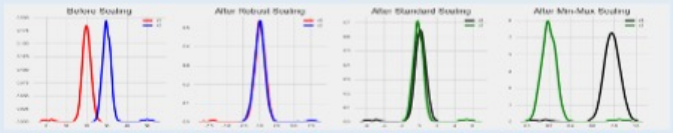
LINEAR TRANFORMATIONS

Each feature/column is transomed separately



Linear tranfs. **preserve** relative distance between features, and the shape of the distribution

IN PRACTICE: we often ignore the shape of the distribution and just transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation.



Original data
2 features with different means, and probably different SD

Robust Scaling
Both features have:
- MEAN = 0
- SD = IQR

Standard Sc.
Both features have:
- MEAN = 0
- SD = 1

Centred around the same value , and expressed with the same unit & SD

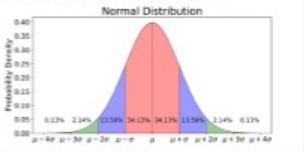
MinMax Scaler
Features have DIFFERENT mean and SD, but
- Min value = 0
- Max value = 1

Preserve data sparsity, and have the same range

SCALING

(i) **Used to Scale features with Gaussian distr**.
(ii) **Applied before using Gradient Descent**,
(iii) Applied before techniques that assume that assume normal distrib. Such as, linear regr., logistic regr. and linear discriminate analysis

z = (x-mean)/sd
sd = sqrt(Σ (x-mean)**2 / N)
- *Range [-∞, +∞]*
- *mean = 0*
- *sd = 1*




```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)

# you may use with_mean=False for sparse data
```

ROBUST SCALER

(i) Gives similar results as the standard scaler, but it based on median & IQR region size, instead of mean and SD.
(ii) Applied for data with **OUTLIERS**.
(iii) Can be used with different IQR size eg 5%, 25%, etc.. That may affect the results, and help scaling data to outliers.

X'i = (x-median)/IQR
- *IQR – difference between 25th and 75 percentiles (can be set as different value)*
- *Range [-∞, +∞]*
- *mean = 0*

```
>>> from sklearn.preprocessing import RobustScaler
>>> scaler = RobustScaler ().fit(X_train)
>>> standardized_X = scaler.transform(X_train)

# set different quintile range
>>> scaler = RobustScaler (quantile_range=(15, 85))
>>> scaler = RobustScaler (quantile_range=(value, 100-value)) # eg. value= 30
```

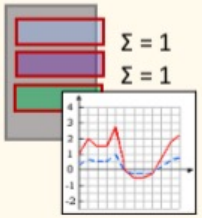
- TO SCALE SPARSE DATA -

NORMALIZATION

Scaling individual samples/rows to unit norm.
-> Sum of values in each observation (row) will be equal to 1 (unit norm).

(ii) Used for **SPARSE DATASETS** with attributes of varying scales
(iii) often used in **text classification and clustering contexts**.
(iv) alg. that use weights, eg NN, and alg, that use distance measures eg. kNN

x/(L1, L2 or Max norm)
- ***Range [0,1]***
- *Sum for each row (norm) = 1*
- ***Smoothing effect***



Normalizer

```
# {'l1', 'l2', 'max'}, default='l2
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer ().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
```

SCALING FEATURES TO A RANGE

Scaling features to lie between a given minimum and maximum value
eg: to convert a temperature from Celsius to Fahrenheit.

- **SPARSE DATA, especially MinMaxScaller** was designed for it.
  - Provide robustness to very small standard deviations of features
  - Do not centre values, and preserve zero entries in sparse data.
- **OUTLIERS**, but in that case there can be a problem (see caution)

Min-Max Scaler

Rescales features between 0-1

(i) Applied before using optimization alg. like **gradient descent**, that assume all values have 0-1 range
(ii) Used with alg. that use weight inputs; regression, NN.
(iii) and with alg. that use distance measures, k-NN

x' = (x – min) / (max – min)
- ***Range [0,1]***
- *If x = min(X), then x'=0*
- *If x = max(X), then x'=1*

CAUTION: may rescale values to very small intervals, if **outliers** are present

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> transformer = MinMaxScaler()
>>> rescaledX = transformer.fit_transform(X)

# you may provide explicirt min/max values
>>> transformer = MinMaxScaler(feature_range=(0, 123))
```

MaxAbs Scaler

Rescales features between -1 & 1
Divides each value by maximum absolute value of each feature.

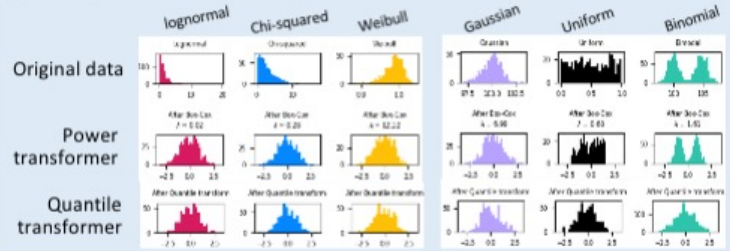
Range [-1,..,1]
NaN - remain NaN

```
>>> from sklearn.preprocessing import MaxAbsScaler
    X = [[ 1., -1., 2.], ... [ 2., 0., 0.], ... [ 0., 1., -1.]]
>>> transformer = MaxAbsScaler().fit(X)
>>> transformer.transform(X)
    array([[ 0.5, -1. , 1. ], [ 1. , 0. , 0. ], [ 0. , 1. , -0.5]])
```

- USED TO CORRECT SKEWNESS IN THE DISTRIBUSIONS -

NON-LINEAR TRANSFORMATIONS

(i) transforms the features to follow a uniform or a normal distribution
(ii) To remove/reduced the impact of **OUTLIERS**
(iii) spread out the most frequent values



CAUTION:
(1). these methods don't scale the data to a predetermined range
(2). These methods may distort lin. corr. between var's measured at the same scale, but renders var's measured at different scales more directly comparable

Quantile transformer

Non-parametric approach

- Range: [0,1]
- spread out the most frequent values
- transformed data is the approximation of the quantile position of the actual data

How it works?
- first, cdf is used to map the original values to a uniform distribution.
- then, these values are mapped to output distribution with quantile function.
- values below or above the fitted range will be mapped to the bounds of the output distr.

```
>>> from sklearn.preprocessing import QuantileTransformer
>>> qt = QuantileTransformer(n_quantiles=10, random_state=0)
>>> qt.fit_transform(X)
```

- **n_quantiles;** Typically large, default 1000.
- **output_distribution** *{'uniform', 'normal'},* distrib. used for the transformed data.
- **ignore_implicit_zeros** If True, zeros are not used, and stay as zeros.
- **Subsample** *int,* max nr of used samples

	mpg	mpg_trans
0	18.0	0.288673
1	15.0	0.153652
2	18.0	0.288673
3	16.0	0.202771
4	17.0	0.238295

Power transformer

Parametric approach

- map data from any distribution to a Gaussian distribution
- stabilize variance
- minimize skewness.

Two methods:
'box-cox' - needs the data to be positive
'Yeo-Johnson' - data to be both negative and positive.

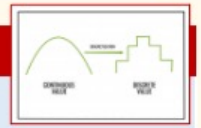
```
>>> pt = PowerTransformer(method='box-cox', standardize=False)
>>> X_lognormal = np.random.RandomState(616).lognormal(size=(3, 3))
>>> pt.fit_transform(X_lognormal)
```

Skilearn API

`fit`(X[, y])	Compute the median and quantiles to be used for scaling.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X)	Scale back the data to the original representation
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Center and scale the data.

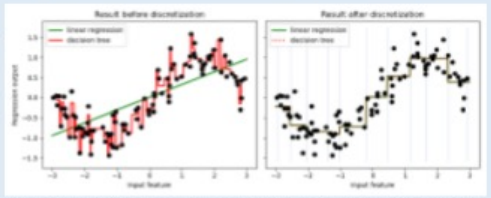
SKLEARN TRANFORMERS – PART 2

DISCRETISATION/BINNING



(i) Used to discretize continuous features
(ii) May improve linear models,
(iii) Reduce variance in non-linear models, like regression trees,

After discretization, linear regression and decision tree make exactly the same prediction



https://scikit-learn.org/stable/auto_examples/preprocessing/plot_discretization.html#sphx-glr-auto-examples-preprocessing-plot-discretization-py

binarization

All values > threshold are 1 and all ≤threshold are marked as 0

(i) **Applied for PROBABILITIES**
(ii) Used for Feature eng.

Generates 0 or 1 values only

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X_train)
```

K-bins discretization

Transforms continuous feature into a categorical feature by partitioning it into several bins within the expected value range (intervals).

- ***0 -K-1 bins***
- ***Ecoding***
  - ***One-hot-enc***
  - ***ordinal (0, 1, 2, …, k-1)***

Value	Bins
10	1
15	1
20	1
25	2
30	2

 # Encoding

- ***'onehot',: default, i***gnored features, eg missing data, novelties, are always stacked to the right
- ***'ordinal':*** return sthe bin identifier encoded as an integer value

CAUTION
Ordinal values may still be used, in most models

 # Strategy used to define the widths of the bins.

- **Uniform;** All bins in each feature have identical widths (max-min values).
- **Quantile;** All bins in each feature have the same number of points.
- **kmeansValues;** in each bin have the same nearest center of a 1D k-means cluster.

```
>>>  from sklearn.preprocessing import KBinsDiscretizer
>>> enc = KBinsDiscretizer(n_bins=10, encode='onehot')
>>> X_binned = enc.fit_transform(X)
```

ENCODING CATEGORICAL FEATURES & TARGETS

one-hot/dummy encoding

Nan & None
- Considered as separate values,

handle_unknown='ignore'
- If ignore, these are mapped to zeros ,

drop='first'
- column with the 1st category is dropped,
- If only one cat is present, it is also dropped,
- Dropped column contains zeros, representing unknown variables (novelties)

Why to drop a column?
Done to avoid co-linearity in the input matrix in some classifiers. Eg., non-regularized regression (LinearRegression), since co-linearity would cause the covariance matrix to be non-invertible

```
>>> from sklearn.preprocessing import OneHotEncoder()
>>> enc = OneHotEncoder(
            drop='first',
            handle_unknown='ignore' )
            # unknown/novelties = will be encoded with zero
>>> enc_X = enc.fit_transform(X)
>>> enc.categories_ # returns categories,

# you may specify explicit categories with 'categories' parameter
- NOT ADVISED
 >>> X =    [['male', 'uses Safari'], ['female', 'uses Firefox']]
 >>> genders  = ['female', 'male']
 >>> browsers  = ['uses Firefox', 'uses Safari']
 >>> enc = OneHotEncoder(categories=[genders, browsers])
```

OrdinalEncoder()

transforms each categorical feature to one new feature of integers (0 to n_categories - 1)

IMPORTANT
Ordinal values can not be used directly with all scikit-learn estimators, as these expect continuous input

 # handle_unknown & unknown_value
- If =='error', an error will be raised in case of novelty is detected.
- If =='use_encoded_value', alg will use value provided with unknown_value, None, will be returned in inverse_transform

 # NaN
- Stays NaN

```
>>> from sklearn.preprocessing import OrdinalEncoder()
>>> enc = OrdinalEncoder()
>>> encoded_X = enc.fit_transform(X_train)
```

get_dummies()

One-hot encoder from pandas

https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html

Time related feature eng

https://scikit-learn.org/stable/auto_examples/applications/plot_cyclical_feature_engineering.html#sphx-glr-auto-examples-applications-plot-cyclical-feature-engineering-py

ADDING POLYNOMIAL FEATURES

PolynomialFeatures()

- **interaction_only**, if True, only the highest degree is used
- **include_bias** *If True, adds bias term, w0, with 1 only in each row.*
  For intercept

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly = PolynomialFeatures(degree=3, include_bias=True)
>>> poly.fit_transform(X)
>>> poly_get_feature_names() # will retunr names of all new features
```

with FunctionTransformer() And labda function

```
>>> poly_columns = ['col_1',  'col_2']
>>> poly_transformer = Pipeline([
            ('scaler', StandardScaler()),
            ('poly', FunctionTransformer(
                lambda X: np.c_[X, X**2, X**3])  ])
```

- **It wont add bias term !**

CUSTOM TRANFORMERS

FunctionTransformer()

Used to implement a transformer from an arbitrary function with transformer API

 # Example 1. build a transformer with a log transformation

```
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p, validate=True)
>>> transformer.fit_transform(X)
    # validate=True; check if the imput is an array

# ensure that inverse_tranform is possible
>>> enc = transformer.fit(X, check_inverse=True)
    # if not, returns a warning
    # Use fit, before transform, to apply it.
```

 # Example 2. provide stats for twitter posts, eg. post length, and sentence nr.

```
>>> def text_stats(posts):
...             return [{'length': len(text),
...                      'num_sentences': text.count('.')}
...                      for text in posts]
>>> text_stats_transformer = FunctionTransformer(text_stats)
```

 # Example 3. just add 1 to each value in a transformed columns

```
>>> def cust_func(x):
....            return x +1
```

LABEL ENCODERS

LabelEncoder()

 # Encodes labels/targets in y,

```
>>> from sklearn.preprocessing import LabelEncoder
>>>  enc = LabelEncoder()
>>>  y = enc.fit_transform(y)
```

LabelBinarizer()

- ***Works, like one-hot encoder,***
- ***Can be used with mutivariate regression***

Sourses

https://towardsdatascience.com/5-data-transformers-to-know-from-scikit-learn-612bc48b8c89
https://scikit-learn.org/stable/modules/preprocessing.html#spline-transformer
https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html#sphx-glr-auto-examples-preprocessing-plot-all-scaling-py

IMPUTATION WITH SKLEARN

MISSING VALUES

DELETION — *Pandas functions/pre-processing*

- **PAIRWISE DELETIONS** — Deleting only missing data in compared cases
- **LIKEWISE DELETIONS** — dropping entire rows, with missing values
- **DROPPING COLUMNS** — dropping entire columns, with missing values

ML alg's require numeric input values, and a value to be present for each row and column in a dataset. It is common to identify missing values and replace them with a numeric value.

IMPUTATION

- **ADVANCED**
  - **kNN Based** — "nearest neighbor imputation" **kNN model** is used to predict the missing values, although any other model can also be applied *IterativeImputer()*
  - **MICE** — MICE; multiple imputation method used to replace missing data **values** in a data set under certain assumptions about the data missingness mechanism (e.g., the data are missing at random, the data are missing completely at random) *KNNImputer()*
- **GENERAL**
  - **Constant**
  - **Non missing = 1, Missing = 0** → *MissingIndicator()*
  - **Mean, Median, Mode, Most Frequent** → *SimpleImputer()*
- **TIME SERIES**
  - **Forward Fill**
  - **BackFill**
  - **Linear Interpolation** → *Pandas functions*

SimpleImputer()

Imputation transformer for completing missing values

- **missing_values;** defining values considered as missing in a dataset
- **Strategy;** *default='mean'*
  - **"mean"/ "median"**
  - **"most_frequent";** works for both, str, and numeric data, Caution, if >1 value, the smallest is used.
  - **"constant";** you must define **fill_value.**
- **add_indicator,** If True, a MissingIndicator transform will stack onto output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation. If a feature has no missing values at fit/train time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

```
>>> fomr sklearn.impute import SimpleImputer
>>> imp = SimpleImputer(
            missing_values=0,
            strategy='mean',
            axis=0)
>>> imp.fit_transform(X_train)
```

KNNimputer()

(i) ***kNN imputation (K-nearest neighbor model)***

- *(i)* *A new sample is imputed by finding the samples in the training set "closest" to it and averages these nearby points to fill in the value.*
- *(ii)* *More efficient then the commonly used row average method or replacing nan with Zeros*

(ii) *Alternatively:* **regression models** can be used for numeric variables,

(iii) Configuration of KNN imputation involves selecting:

- (iii) **distance measure** (e.g. Euclidean)
- **(iv) k hyperparameter,** ie. the number of contributing neighbors for each prediction

```
>>> from sklearn.impute import KNNImputer
>>> imputer = KNNImputer(
...         n_neighbors=5,
...         weights='uniform',
...         metric='nan_euclidean')
...                 # nan avare method, do not use nan to compute distances,
...                 # if other methods – you must provide distance matrices,
...                   or your function
>>> imputer.fit(X)
>>> Xtrans = imputer.transform(X)
```

GOOD PRACTICE

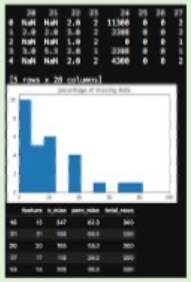
Convert missing values to np.nan in pandas.df that may have pd.NA

Example: Tuning KNNinputer

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from numpy import isnan
from sklearn.impute import KNNImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
```

1. Load data, EDA on missing data points

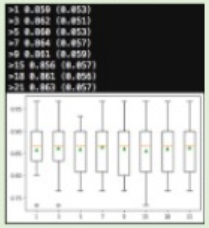
```
# load dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/horse-colic.csv'
dataframe = pd.read_csv(url, header=None, na_values='?')

# summarize the first few rows
print(dataframe.head())

# summarize the number of rows with missing values for each column
results=[]
for i in range(dataframe.shape[1]):
    # count number of rows with missing values
    n_miss = int(dataframe[[i]].isnull().sum())
    perc = float(n_miss/dataframe.shape[0]*100)
    results.append({"feature":i, "n_miss":n_miss,
                    "perc_miss":np.round(perc,1),
                    "total_rows":dataframe.shape[0]})

# show all features on histogram
fig, ax = plt.subplots();
ax.hist(pd.DataFrame(results).perc_miss.values)
ax.set_xlim(0,100)
ax.set_title("pecentage of missing data")
plt.show();

# display columns with the highest pers/nr of missing values
pd.DataFrame(results).sort_values("perc_miss", ascending=False).head()
```



2. Find best k, suing k-fold cv, and random forest classifier

```
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]

# define modeling pipeline
model = RandomForestClassifier()
imputer = KNNImputer()

# evaluate each strategy on the dataset
results = list()
strategies = [str(i) for i in [1,3,5,7,9,15,18,21]]  # k for KNNimputer
for s in strategies:
    # create the modeling pipeline
    pipeline = Pipeline(steps=[
        ('i', KNNImputer(n_neighbors=int(s))),
        ('m', RandomForestClassifier())
    ])

    # evaluate the model
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)

    # store results
    results.append(scores)
    print('>%s %.3f (%.3f)' % (s, np.mean(scores), np.std(scores)))

# plot model performance for comparison
plt.boxplot(results, labels=strategies, showmeans=True)
plt.show()
```

```
>1 0.859 (0.053)
>3 0.862 (0.051)
>5 0.860 (0.053)
>7 0.864 (0.057)
>9 0.861 (0.059)
>15 0.856 (0.057)
>18 0.861 (0.056)
>21 0.863 (0.057)
```



3. Create pipeline with selected steps& hyperparameters, fit it, and use for prediction for new data

```
# define new data
row = [2, 1, 530101, 38.50, 66, 28, 3, 3,
       np.nan, 2, 5, 4, np.nan, np.nan, np.nan,
       3, 5, 45.00, 8.40, np.nan, np.nan,
       2, 11300, 00000, 00000, 2]

# create the modeling pipeline
pipeline = Pipeline(steps=[
    ('i', KNNImputer(n_neighbors=21)),
    ('m', RandomForestClassifier())
])

# fit the model
pipeline.fit(X, y)

# make a prediction
yhat = pipeline.predict([row])
print('Predicted Class: %d' % yhat[0])
```

```
Predicted Class: 2
```

The plot suggest that there is not much difference in the k value when imputing the missing values, with minor fluctuations around the mean performance (green triangle).

Example taken from: https://machinelearningmastery.com

Data Preprocessing

APPLYING MIXED TYPE TRANSFORMERS

SimpleAI.ch

THE SIMPLEST PIEPLINE WITH MIXED TRANFORMER TYPES

```
# Author: Pedro Morales <part.morales@gmail.com>
#
# License: BSD 3 clause

from __future__ import print_function

import pandas as pd
import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV

np.random.seed(0)

# Read data from Titanic dataset.
titanic_url = ('https://raw.githubusercontent.com/amueller/'
               'scipy-2017-sklearn/091d371/notebooks/datasets/titanic3.csv')
data = pd.read_csv(titanic_url)

# We will train our classifier with the following features:
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression(solver='lbfgs'))])

X = data.drop('survived', axis=1)
y = data['survived']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

1. **Define names of eg. categorical & numeric features**
   - *List[] with feature names*
2. **Define tranformation steps for each feature type**
   - *Pipepline(steps=[])*
   - *make_pipeline()*
   - *FunctionTransformer()*
3. **Create Preprocessor With mini-pipelines for different feature types**
   - *ColumnTranformer()*
4. **Add preprocessor to larger pipeline, that also contains the model to fit**
   - *Pipepline(steps=[(“prep”, preprocessor()), (“clasif”, ml_alg_name())])*
5. **Prepare data (train/test split) Fit the model & Evaluate it**
   - *X,y = input_data*
   - *train_test_split(x,y , test_size=0.3)*
   - *Pipeline.fit(x_train, y_train)*
   - *Pipeline.score(X_test, y_Test)*
   - *Piepline.predict(X_test, y_Test)*

Similar example, provided by skilearn with some options for automated interactions with input data in pandas dataframe

Column Transformer with Mixed Types

This example illustrates how to apply different preprocessing and feature extraction pipelines to different subsets of features, using `ColumnTransformer`. This is particularly handy for the case of datasets that contain heterogeneous data types, since we may want to scale the numeric features and one-hot encode the categorical ones.

In this example, the numeric data is standard-scaled after mean-imputation, while the categorical data is one-hot encoded after imputing missing values with a new category (`'missing'`).

In addition, we show two different ways to dispatch the columns to the particular pre-processor: by column names and by column data types.

Finally, the preprocessing pipeline is integrated in a full prediction pipeline using `Pipeline`, together with a simple classification model.

```
# Author: Pedro Morales <part.morales@gmail.com>
#
# License: BSD 3 clause

import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.datasets import fetch_openml
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV

np.random.seed(0)

# Load data from https://www.openml.org/d/40945
X, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)

# Alternatively X and y can be obtained directly from the frame attributes:
# X = titanic.drop('survived', axis=1)
# y = titanic['survived']
```

Use ColumnTransformer by selecting column by names

We will train our classifier with the following features:

Numeric Features:

- age: float;
- fare: float.

Categorical Features:

- embarked: categories encoded as strings {'C', 'S', 'Q'};
- sex: categories encoded as strings {'female', 'male'};
- pclass: ordinal integers {1, 2, 3}.

We create the preprocessing pipelines for both numeric and categorical data. Note that `pclass` could either be treated as a categorical or numeric feature.

```
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression())])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=0)

clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

```
Out: model score: 0.790
```

Defining dtypes for mixed tranformers automatically

Select columns in pd.DataFrame by their dtypes using skilearn selector

df.info()

Used to fiund out how the columns will be categorized with the selecto

make_column_selector()

Can select columns based on

(i) Pandas data frame dtypes

(ii) or the columns name with a regex.

CAUTION: When using multiple selection criteria, **all** criteria must match for a column to be selected.

add i. supported Pandas df dtypes:

- ***numeric:*** *use “numeric”* , np.number, 'number'
- **categorical:** use “category”
- **object:** use object - this included str, and all other columns encoded as object (may be of any dtype)
- **datetimes:** use np.datetime64, 'datetime' or 'datetime64’
- **timedeltas:** use np.timedelta64, 'timedelta' or 'timedelta64’
- **datetimetz:** use 'datetimetz’, or 'datetime64[ns, tz]'

```
>>> from sklearn.compose
...     import make_column_selector
...     as selector
>>> preprocessor = ColumnTransformer([
....     (num, numeric_transformer,
....     Selector(dtype_include=“Category”),
```

MAPRAMETERS:

- **Pattern**
- **dtype_include**
- **dtype_exclude**

Example with selecting categorical and numerical dtypes

Use ColumnTransformer by selecting column by data types

When dealing with a cleaned dataset, the preprocessing can be automatic by using the data types of the column to decide whether to treat a column as a numerical or categorical feature. `sklearn.compose.make_column_selector` gives this possibility. First, let's only select a subset of columns to simplify our example.

```
subset_feature = ['embarked', 'sex', 'pclass', 'age', 'fare']
X_train, X_test = X_train[subset_feature], X_test[subset_feature]
```

Then, we introspect the information regarding each column data type.

```
X_train.info()
```

```
Out: <class 'pandas.core.frame.DataFrame'>
     Int64Index: 1047 entries, 1118 to 684
     Data columns (total 5 columns):
      #   Column    Non-Null Count  Dtype
     ---  ------    --------------  -----
      0   embarked  1045 non-null   category
      1   sex       1047 non-null   category
      2   pclass    1047 non-null   float64
      3   age       841 non-null    float64
      4   fare      1046 non-null   float64
     dtypes: category(2), float64(3)
     memory usage: 35.0 KB
```

We can observe that the `embarked` and `sex` columns were tagged as `category` columns when loading the data with `fetch_openml`. Therefore, we can use this information to dispatch the categorical columns to the `categorical_transformer` and the remaining columns to the `numerical_transformer`.

Note: In practice, you will have to handle yourself the column data type. If you want some columns to be considered as `category`, you will have to convert them into categorical columns. If you are using pandas, you can refer to their documentation regarding Categorical data.

```
from sklearn.compose import make_column_selector as selector

preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, selector(dtype_exclude="category")),
    ('cat', categorical_transformer, selector(dtype_include="category"))
])
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression())])


clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

```
Out: model score: 0.794
```

The resulting score is not exactly the same as the one from the previous pipeline because the dtype-based selector treats the `pclass` column as a numeric feature instead of a categorical feature as previously:

```
selector(dtype_exclude="category")(X_train)
```

```
Out: ['pclass', 'age', 'fare']
```

```
selector(dtype_include="category")(X_train)
```

```
Out: ['embarked', 'sex']
```

HOW TO USE DIFFERENT TRANSFORMERS FOR DIFFERENT COLUMNS

sparse matrices vs numpy arrays
https://stackoverflow.com/questions/36969886/using-a-sparse-matrix-versus-numpy-array

HOW TO VISUALIZE THE PIPELINE

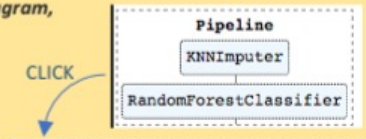
set_configs()

Use function that set global scikit-learn configuration And ask for displaying nice diagram, instead of text

CAUTION: it changes all global settings!

```
>>> from sklearn import set_config
>>> set_config(display="diagram")
    # “text” is an alternative, and
      normally used.
>>> pipeline
```

CLICK




```
Pipeline
Pipeline(steps=[('i', KNNImputer(n_neighbors=21)),
                ('m', RandomForestClassifier())])
```

```
KNNImputer
KNNImputer(n_neighbors=21)
```

```
RandomForestClassifier
RandomForestClassifier()
```

APPLYING CUSTOM TRANSFORMERS

EXAMPLE: APPLYING CUSTOM TRANFORMERS

```
>>> import numpy as np
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import FunctionTransformer
>>> from sklearn.preprocessing import OneHotEncoder, StandardScaler, KBinsDiscretizer
>>> from sklearn.compose import ColumnTransformer

# 1. DEFINE CUTOM TRANFROMERS
# create custom transformer
'' unlike pipeline, make_pipepline
  generates names for steps automatically
  ->> lowercase name of an estimator
  otherwise, these two fucntions work in the same way
'''
>>> log_scale_transformer = make_pipeline(
...                         FunctionTransformer(np.log, validate=False),
...                         StandardScaler()
...                         )

# 2. DEFINE PREPROCESSING FUNCTION WITH DIFFERENT  TRANFORMATIONS
      FOR DIFFERENT COLUMNS
# Create final preprocessor for the data, with ColumnTransformer()
''' ColumnTransformer takes a list with instrucitons for each column/col. Group.
   for each of them you must provide the folloiwing
            (i)  unique preprocessor name
            (ii). Transformer function, "passthrough" str, to not make any modifications
                        - if more then one function, you need to create it separately,
                          using pieline or make pipline,
                          like log_scale_transformer in this example.
            (iii) column names in input data, that will be transformed  (LIST)
'''
linear_model_preprocessor = ColumnTransformer(
   transformers=[

      ("passthrough_numeric", "passthrough", ["column_1"]),

      ("binned_numeric", KBinsDiscretizer(n_bins=10), [" column_2", " column_ 3"]),

      ("log_scaled_numeric", log_scale_transformer, [" column_4"]),

      ("onehot_categorical", OneHotEncoder(), [" column_5", " column_5", " column_6"]),
   ],
   remainder="drop",     # TWO OPTION {'drop', 'passthrough'}
)
```

set_config(display="diagram")

When to use pipeline vs make_pipeline?

With Pipeline:
- Step/tranformer names are explicit,
- Ie. the name wotn't change if you change estimator/transformer used in a step,
  - e.g. if you replace LogisticRegression() with LinearSVC() you can still use clf__C.

With make_pipeline:
- shorter in use
- names are auto-generated using a straightforward rule (lowercase name of an estimator).
- Cons: if you change the function, names inside will change, an some other functions may stop working in the pipeline,

Thus:
- Use make_pipeline for quick experiments and
- Use Pipeline for more stable code/larger project
- not a big deal to use make_pipeline/Pipeline interchangeably

make_pipeline()

Here used to create multistep transformer
- Give names to steps automatically,
- Useful for pre-processing steps

FunctionTransformer()

Provides skilearn transformer API to custom functions
- ie. fit(), trasnform(), etc ...
- Allows using custom function with pipeline() & make_pieline to build more complex, mutistep tranformers/pieplines

HOW TO USE IT?

ColumnTransformer()

for: MIXED TRANFORMER TYPES

Allows applying different transformers to different columns in one dataset
- List with tranformer + column names (in a list)
- Option to leave some columns without chnages,
  - See, **"passthrough"**
- Option to remove, other columns,
  - See, **"drop"**

HOW TO BUILD CUSTOM TRANFORMERS

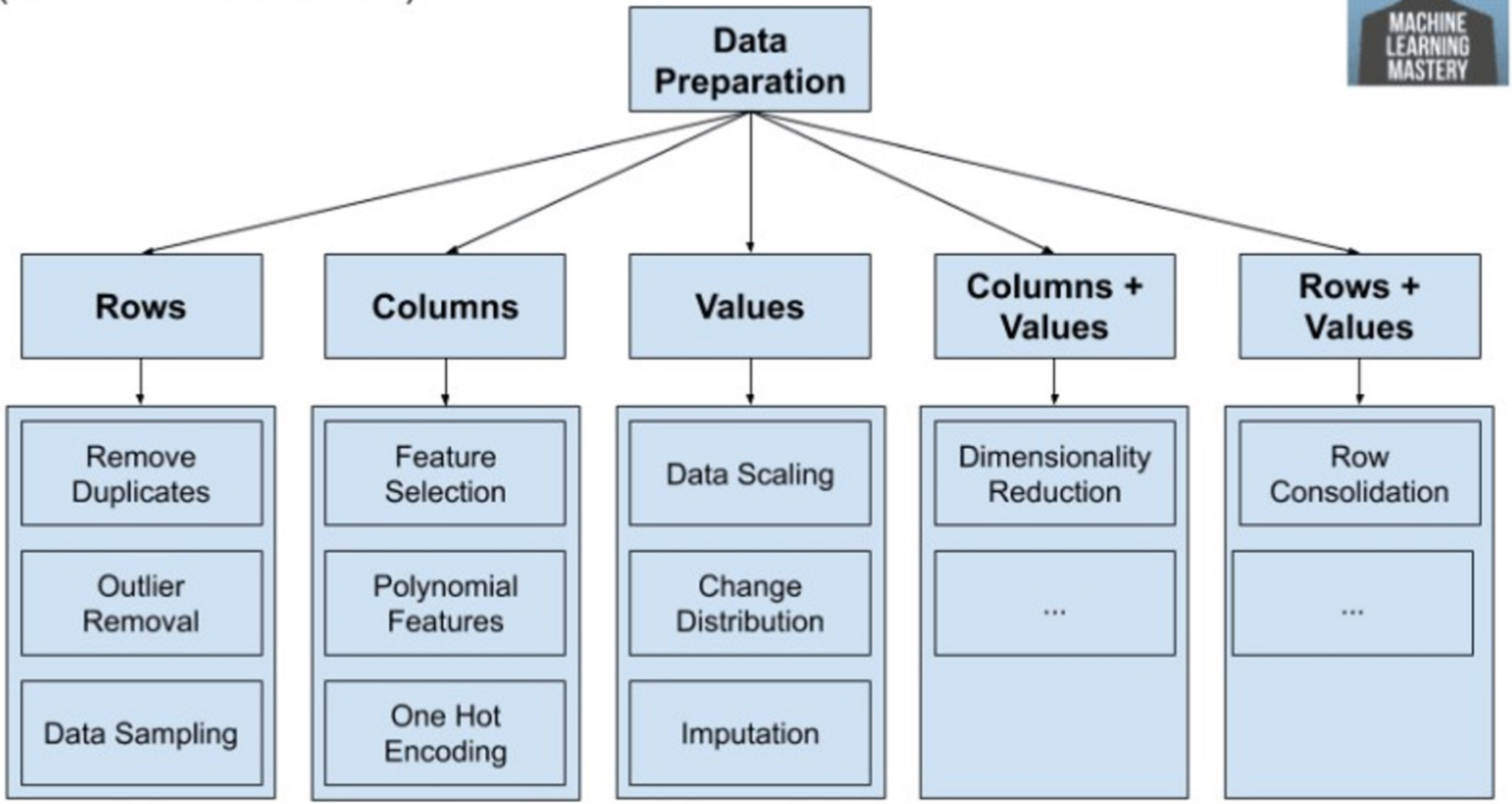
HOW TO USE IT?

```
# Example 1.  build a transformer with a log transformation
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p, validate=True)
>>> transformer.fit_transform(X)
    # validate=True; check if the imput is an array

# ensure that inverse_tranform is possible
>>> enc = transformer.fit(X, check_inverse=True)
    # if not, returns a warning
    # Use fit, before transform, to apply it.

# Example 2. provide stats for twitter posts, eg. post length, and sentence nr.
>>> def text_stats(posts):
...         return [{'length': len(text),
...                  'num_sentences': text.count('.')}
...                  for text in posts]
>>> text_stats_transformer = FunctionTransformer(text_stats)

# Example 3. just add 1 to each value in a transformed columns
>>> def cust_func(x):
....        return x +1
```

SMALL PIPELINE WITH ESTIMATOR

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score

# load the data
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target

# split to train/test
>>> X_train, X_test, y_train, y_test = train_test_split(
                                X, y, train_size=0.7, random_state=0)

# scale input data
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)

# create classifier
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)

# predict & test
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

Data Preparation Framework
(*for structured/tabular data*)



Feature Space & Dimensionality reduction

FEATURE ENG. - OVERVIEW

Feature engineering techniques

Numerical Range
- Scaling
- Normalizing
- Standardizing

Grouping
- Bucketizing
- Bag of words

Other techniques

Dimensionality reduction in embeddings
- Principal component analysis (PCA)
- t-Distributed stochastic neighbor embedding (t-SNE)
- Uniform manifold approximation and projection (UMAP)

Feature crosses
- Combines multiple features together into a new feature
- Encodes nonlinearity in the feature space, or encodes

Online tools

- Intuitive exploration of high-dimensional data
- Visualize & analyze
- Techniques
  - PCA
  - t-SNE
  - UMAP
  - Custom linear projections
- Ready to play

Ready to play

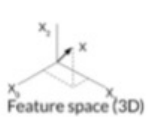
@ projector.tensorflow.org

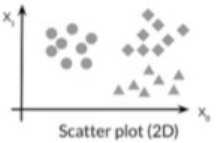
FEATURE SPACE

- N dimensional space defined by your N features
- Not including the target label

$$X = \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_d \end{bmatrix}$$

Feature vector





Feature space

3D Feature Space

No. of Rooms X_0	Area X_1	Locality X_2	Price Y
5	1200 sq. ft	New York	$40,000
6	1800 sq. ft	Texas	$30,000

$Y = f(X_0, X_1, X_2)$

f is your ML model acting on feature space X_0, X_1, X_2

DIMENSIONALITY REDUCTION

- EDA
  - to test whether there is no batch effect
  - to find hidden clusters,
- Feature – extraction
  - Dimensionality reduction
  - To speed up the computation
  - simplifies the model
- Supervised ML:
  - to reduce overfitting/variance
  - increase the accuracy, of the model
- For feature selection

PCA

The idea behind PCA:
- PCA reduces the nr. of dimensions by projecting the data onto a set of n-ortogonal axes (90degr. to each other), called the PRINCIPIAL COMPONENTS. If we choose the set of "good axes" we can reduce the loss of information
- How: we create ne axes om a plot and project the overall variability in data from all dimensions in relation to that axis. A set of orthogonal axes with min. variance is selected.
- Good for analysis of high-dimensional data
- CAUTION: PCA alg. Favours features with the highest variance
- IMPORTANT: max incompetents == nr. Of features in the dataset.

Feature Granularity

Transformations	
Instance-level	**Full-pass**
Clipping	Minimax
Multiplying	Standard scaling
Expanding features	Bucketizing
etc.	etc.

Ensure feature space coverage !

- Train/Eval datasets representative of the serving dataset
  - Same numerical ranges
  - Same classes
  - Similar characteristics for image data
  - Similar vocabulary, syntax, and semantics for NLP data
- Data affected by: seasonality, trend, drift.
- Serving data: new values in features and labels.
- Continuous monitoring, key for success!

UMAP

- UMAP is a non-linear method for dimensionality reduction
- IDEA:
  - The dimensionality of many datasets is only artificial y high, and the data in these datasets depends on smalled number of confounding factors
  - UMAP alg. Is based on nearest neightbours alg.

When to transform the data

1. we can prepare entire dataset and then build the model

Pre-processing training dataset

Pros	Cons
Run-once	Transformations reproduced at serving
Compute on entire dataset	Slower iterations

2. we can do tranformation withitnn the model

Transforming within the model

Pros	Cons
Easy iterations	Expensive transforms
Transformation guarantees	Long model latency
	Transformations per batch: skew

BOX: Normalization vs Standarization

Normalization vs. standardization
- Normalization is good to use when:
  - the distribution of your data does not follow a Gaussian distribution.
  - This can be useful in algorithms that do not assume any distribution of the data like K-Nearest Neighbors and Neural Networks.
- Standardization,can be helpful:
  - the data follows a Gaussian distribution.
  - However, this does not have to be necessarily true.
  - Unlike normalization, standardization does not have a bounding range. So, even if you have **outliers** in your data, they will not be affected by standardization.
- You can always start by fitting your model to raw, normalized and standardized data and compare the performance for best results.
- Scaling of target values is generally not required.
- based on: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/

T-SNE

UUSED FOR:
- EDA
- Visualization of high-dimensional data

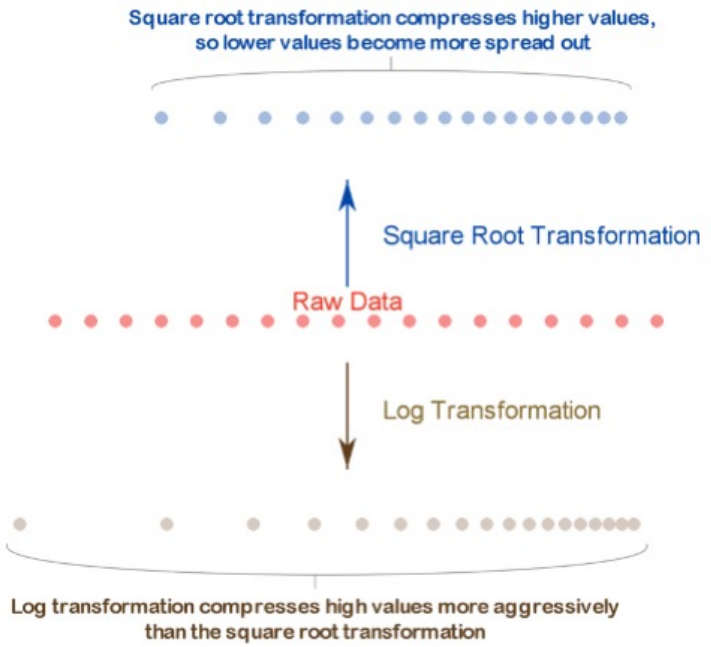
The idea behin t-SNE
Converts similarities between data points into joint-probabilities Tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high dimensional data - In many cases t-SNE is the best method to visualize the dataset

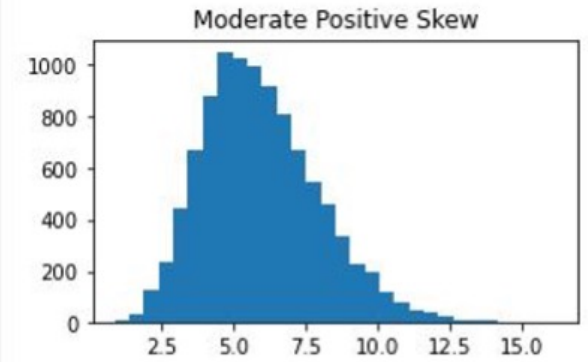
Problems with t-SNE
- t-SNE had non-convex cost function == with different initialization points we will get different results. Therefore, if is best to use other dimensionality reduction method, eg
  - PCA for dense data
  - TruncatedSVD for sparse data
- Time consuming
  - you may reduce the nr. Dimensions with PCA and then use t-SNE
- **LIMITED TO FIT_TRANSFOMR**
  - t-.SNE an only be used to visualized or prepare the test data, but not to transform test data
  - Can not be used as data pre-processing step !

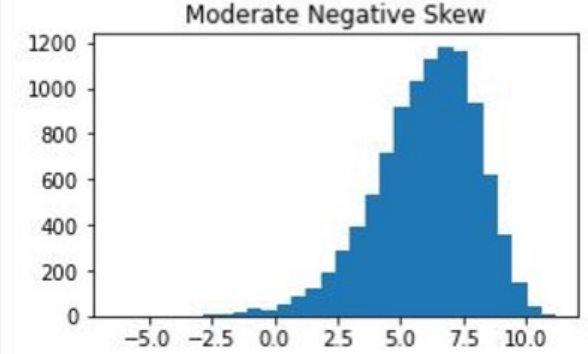
NON-LENEAR FEATURE TRANFORMATIONS

SOLUTION 1. FEATURE TRANSFORMATIONS

- **Scaling & shift (f(x)±c) –not very helpful,**
- **Reflection** (-f(x), resulting the same function but on the other site of the axis),
- **log transformation** (log(x), ln(x), used for right-skewed distribusions),
  - Data skewed to the right (i.e. in the positive direction).
  - The residual's standard deviation is proportional to fitted values
  - The data's relationship is close to an exponential model
  - the residuals reflect multiplicative errors that have accumulated during each step of the computation
- **Sqrt transformations;** it basically makes a straight line from power fn line, and its used. Sqrt transf. compresses larger values, and makes differences between small values more apparent, but it is made less agressively then log transf.
- **power transformation** (eg: x^2), used when The data's relationship is close to an exponential model
- **repricotal transformation** (1/x, - dramatic effect on the shape of the distribution, reversing the order of values with the same sign. The transformation can only be used for non-zero values.),
- **Share mapping** (all points along one line stay fixed, while other points are shifted parallel to the line by a distance proportional to their perpendicular distance from the line)
- More info and some text has been taken from: https://www.calculushowto.com/transformations/







https://quantifyinghealth.com/square-root-transformation/

Transforming non-norma

$$y = \beta_0 e^{\beta_1 x} \qquad y' = ln(y)$$

$$y = \beta_0 + \beta_1 log(x) \qquad x' = log(x)$$

$$y = \frac{x}{\beta_0 x - \beta_1} \qquad y' = \frac{1}{y}, x' = \frac{1}{x}$$

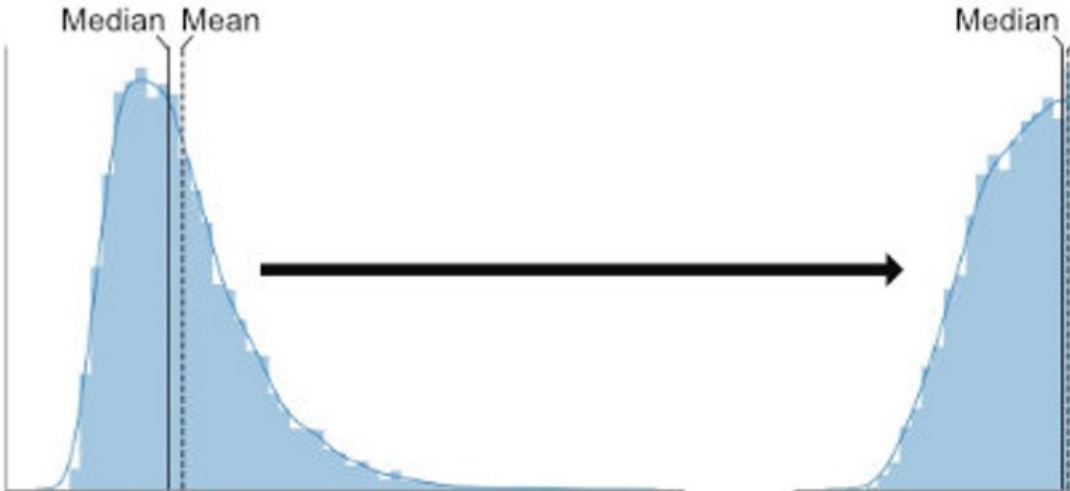
Resulti

$y' = log$

$y' = ln($

$y = \beta_0$ -

$y' = \beta_0$



Q. What is the Feature

It is an individual measurable property or characteristic of a phenomenon. Tabular data is described in terms of observations or instances (rows) that are made up of variables or attributes (columns). An attribute could be a feature.

Q. Why it is important?

- **Better features means better results**.;
- **Better features means simpler models;** With good features, you are closer to the underlying problem. You do not need to work as hard to pick the right models and the most optimized parameters.
- **Better features means flexibility; good features** allow you to use less complex models that are faster to run, easier to understand and easier to maintain

Q. Observations vs features

- In computer vision, an image is an observation, but a feature could be a line in the image.
- In MLP, a document or a tweet could be an observation, and a phrase or word count could be a feature.
- In speech recognition, an utterance could be an observation, but a feature might be a single word or phoneme.

Q. Feature is not always an attribute(column)

A feature is an attribute that is useful or meaningful to your problem. It is an important part of an observation for learning about the structure of the problem that is being modeled.

Q. What is Feature Engineering?

i) *Feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data.*

ii) *feature engineering is manually designing what the input x's should be*

Q. What questions you need to ask?

i) what is the best representation of the sample data to learn a solution to your problem?; ie *you have to turn your inputs into things the algorithm can understand*

ii) *How can I decompose or aggregate raw data to better describe the underlying problem?*

Q. What feature engineering you do may depend on:

- The performance measures you've chosen (RMSE? AUC?)
- The framing of the problem (classification? regression?)
- The predictive models you're using (SVM?)
- The raw data you have selected and prepared (samples? formatting? cleaning

Feature Engineering Basics

Feature engineering means building additional features out of existing data which is often spread across multiple related tables. Feature engineering requires extracting the relevant information from the data and getting it into a single table which can then be used to train a machine learning model. The process of constructing features is very time-consuming because each new feature usually requires several steps to build, especially when using information from more than one table. We can group the operations of feature creation into two categories: **transformations** and **aggregations**. Let's look at a few examples to see these concepts in action.

AUTOMATED FEATURE ENGINEERING

SimpleAI.ch

Feature Engineering

building additional features out of existing data which is often spread across multiple related tables.

extracting the relevant information from the data and getting it into a single table which can then be used to train a machine learning model.

Main operations

Transformations;
creating new features out of one or more columns in a single table.
Example: log of each value in one column

Aggregations;
performed across tables, and use a one-to-many relationship to group observations and then calculate statistics.

Example: ***Calculating credit score with information collected in different tables in bank databases***

Table one:
- Contains info on each client
- Address, income, work, education etc…
- One client == one row

Table two:
- contains info on loads from clients,
- One client == multiple rows
- We can calculate aggregated stats for each client, eg min, max, mean

Finally you merge the the info in one table,

Featuretools

https://featuretools.alteryx.com

An open source python library for automated feature engineering

- **Entities** == one table/DataFrame
- **EntitySet;** collection of tables and the relationships between them. == data structure of the higher level
- **Feature Primitives**
  - Basic operations used to create new features
  - Featuretools provides large number of transformations and aggregations,
  - Custom features may also be created
- Library has extensions for NLP and Time series (not sure they are ready)
- License: you may use it for commercial purposes, but you cant say it was used in your product without prior authors permission
- Not available in anaconda, but easy to instal via pip or conda forge

Example:

Featuretools

We start by creating an empty entityset in featuretools

```
>>> import featuretools as ft
>>> es = ft.EntitySet(id = 'clients')
```

STEP 1. add entities to entityset

```
# now you need to add entities to entityset
'''each entity must have unique index == column with only unique values'''
>>> es = es.entity_from_dataframe(
            entity_id = 'clients',     # entity id in the entityset
            dataframe = clients,     # source data,
            index = 'client_id',       # index column in the source data
            time_index = 'joined'   # time data index in the added entity,
            )

# add another entity to entityset
'''problem: 1. no unique index
            2. column 'missed' may obtain wrong dtype
               thus we must specify one (see below) '''
>>> es = es.entity_from_dataframe(
            entity_id = 'payments',
            dataframe = payments,
            variable_types = {'missed': ft.variable_types.Categorical},
            make_index = True,        # creates index column with unique values
            index = 'payment_id',     # name of the index column
            time_index = 'payment_date'
            )
```

 # check added values

```
In [16]: es['payments']

Out[16]: Entity: payments
           Variables:
             loan_id (dtype: numeric)
             payment_amount (dtype: numeric)
             payment_date (dtype: datetime_time_index)
             missed (dtype: categorical)
             payment_id (dtype: index)
           Shape:
             (Rows: 3456, Columns: 5)
```

STEP 2. Add relationships between entities

''' we need to specify the variables that link two tables together, eg:
- clients & loans tables == linked via the client_id
- loans & payments == linked with the loan_id'''

```
# Relationship between clients and previous loans
>>> r_client_previous = ft.Relationship(
            es['clients']['client_id'],
            es['loans']['client_id']
            )
>>> r_payments = ft.Relationship(
            es['loans']['loan_id'],
            es['payments']['loan_id']
            )

# Add the relationships to the entity set
>>> es = es.add_relationship(r_client_previous)
>>> es = es.add_relationship(r_payments)
>>> es
'''entityset now contains three entities
  and two relationships'''
```

```
Entityset: clients
  Entities:
    clients [Rows: 25, Columns: 6]
    loans [Rows: 443, Columns: 8]
    payments [Rows: 3456, Columns: 5]
  Relationships:
    loans.client_id -> clients.client_id
    payments.loan_id -> loans.loan_id
```

deep feature synthesis

- Feature primates are automatically generated in all possible combinations
- These primitives can be used by themselves or combined to create features.

```
# MANUALLY DEFINE NEW FEATURES

""Create new features using specified primitivesfeatures,""
>>> feature_names = ft.dfs(
            entityset = es,
            target_entity = 'clients',   # table where we want to add the features
            agg_primitives = ['mean', 'max', 'percent_true', 'last'],
            trans_primitives = ['years', 'month', 'subtract', 'divide']
            )
'''Result: dataframe of new features for each client (because we made clients
the ….). For example, we have the month each client joined which is a
transformation feature primitive, or a number of aggregation primitives such as
the average payment amounts for each client'''
```

client_id	MONTH(joined)
25707	10
26326	5
26695	8
26945	11
29841	8

client_id	MEAN(payments.payment_amount)
25707	1178.552795
26326	1166.736842
26695	1207.433824
26945	1109.473214
29841	1439.433333

```
# AUTOMATED FEATURE GENERATION
""Create new features using specified primitivesfeatures,""
>>> features, feature_names = ft.dfs(
            entityset=es,
            target_entity='clients',
            max_depth = 2)
            '' how many combinations of feature primitives
                to generate, eg a, b, and a*b, a/b etc…'
            we can stack features to any depth we want ''
>>> Features. Head()
```

client_id	SUM(loans.loan_amount)	SUM(loans.rate)	STD(loans.loan_amount)	STD(loans.rate)	MAX(loans.loan_amount)	MAX(loans.rate)	SKEW(loans.loan_amount)
25707	159279	69.54	4844.418728	2.421285	13913	9.44	-0.172874
26326	116321	40.28	4254.149422	1.501819	13464	6.73	0.135246
26695	143845	44.39	4870.228493	1.517860	14865	6.51	0.154467
26945	116889	42.83	4389.555057	1.564796	14503	5.65	0.156534
29841	179634	62.01	4890.630609	2.663892	14837	6.76	-0.212397

Curse of dimensionality

This approach will quickly generate hundreds of features that we may now need to select from to built a good and simple model

- . PCA
- Correlation
- feature selection methods

Example taken from
https://towardsdatascience.com/automated-feature-engineering-in-python-99baf11cc219